Avaliadores Meta-Circulares

António Menezes Leitão

28 de Maio de 2008

Avaliadores Meta-Circulares

Problema

- Nem sempre uma linguagem de programação facilita a descrição de um problema.

Solução: Abstracção Linguística

- Inventa-se uma nova linguagem.

Avaliador

Avaliador

Um avaliador (ou interpretador) de uma linguagem é um procedimento que, quando aplicado a uma expressão dessa linguagem realiza as operações descritas nessa expressão.

Ideia Fundamental

Um avaliador determina o significado de um programa mas é ele também um programa.

Avaliador Meta-Circular

É um avaliador definido na mesma linguagem que ele avalia.

Avaliador

Avaliador de Scheme

```
(define (eval exp env)
```

Avaliador

Avaliador de Scheme

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
```

Avaliador

Avaliador de Scheme

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))
```

Avaliador

Avaliador de Scheme

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))









        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Avaliador de Scheme

```
(define (self-evaluating? exp)
```

Avaliador

Avaliador de Scheme

```
(define (self-evaluating? exp)
  (cond ((number? exp) #t)
        ((string? exp) #t)
        (else #f)))
```

Avaliador

Avaliador de Scheme

```
(define (self-evaluating? exp)
  (cond ((number? exp) #t)
        ((string? exp) #t)
        (else #f)))


(define (quoted? exp)
```

Avaliador

Avaliador de Scheme

```
(define (self-evaluating? exp)
  (cond ((number? exp) #t)
        ((string? exp) #t)
        (else #f)))


(define (quoted? exp)
  (tagged-list? exp 'quote))
```

Avaliador

Avaliador de Scheme

```
(define (self-evaluating? exp)
  (cond ((number? exp) #t)
        ((string? exp) #t)
        (else #f)))


(define (quoted? exp)
  (tagged-list? exp 'quote))




(define (tagged-list? exp tag)
```

Avaliador

Avaliador de Scheme

```
(define (self-evaluating? exp)
  (cond ((number? exp) #t)
        ((string? exp) #t)
        (else #f)))


(define (quoted? exp)
  (tagged-list? exp 'quote))




(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      #f))
```

Avaliador

Avaliador de Scheme

```
(define (self-evaluating? exp)
  (cond ((number? exp) #t)
        ((string? exp) #t)
        (else #f)))


(define (quoted? exp)
  (tagged-list? exp 'quote))

(define (text-of-quotation exp) (cadr exp))


(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      #f))
```

Avaliador

Selecção

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))









        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Selecção

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))


        ((if? exp) (eval-if exp env))









        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Selecção

```
(define (if? exp)
```

Avaliador

Selecção

```
(define (if? exp)
  (tagged-list? exp 'if))
```

Avaliador

Selecção

```
(define (if? exp)
  (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))
```

Avaliador

Selecção

```
(define (if? exp)
  (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))
```

Avaliador

Selecção

```
(define (if? exp)
  (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
```

Avaliador

Selecção

```
(define (if? exp)
  (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
```

Avaliador

Selecção

```
(define (if? exp)
  (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))
```

Avaliador

Selecção

```
(define (if? exp)
  (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))

(define (make-if predicate consequent alternative)
```

Avaliador

Selecção

```
(define (if? exp)
  (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))

(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))
```

Avaliador

Selecção

```
(define (if? exp)
  (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))

(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))

(define (eval-if exp env)
```

Avaliador

Selecção

```
(define (if? exp)
  (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))

(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))

(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
```

Avaliador

Selecção

```
(define (if? exp)
  (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))

(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))

(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
```

Avaliador

Selecção

```
(define (if? exp)
  (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))

(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))

(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Avaliador

Selecção

```
(define false #f)

(define true #t)
```

Avaliador

Selecção

```
(define false #f)

(define true #t)

(define (true? x)
  (not (eq? x false)))
```

Avaliador

Selecção

```
(define false #f)

(define true #t)

(define (true? x)
  (not (eq? x false)))

(define (false? x)
  (eq? x false))
```

Avaliador

Sequenciação

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))


        ((if? exp) (eval-if exp env))










        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Sequenciação

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))


        ((if? exp) (eval-if exp env))




        ((begin? exp)





        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Sequenciação

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))


        ((if? exp) (eval-if exp env))




        ((begin? exp)
         (eval-sequence (begin-actions exp) env))




        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Sequenciação

```
(define (begin? exp)
```

Avaliador

Sequenciação

```
(define (begin? exp)
  (tagged-list? exp 'begin))
```

Avaliador

Sequenciação

```
(define (begin? exp)
  (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))
```

Avaliador

Sequenciação

```
(define (begin? exp)
  (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))
```

Avaliador

Sequenciação

```
(define (begin? exp)
  (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))

(define (first-exp seq) (car seq))
```

Avaliador

Sequenciação

```
(define (begin? exp)
  (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))
```

Avaliador

Sequenciação

```
(define (begin? exp)
  (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))

(define (make-begin seq) (cons 'begin seq))
```

Avaliador

Sequenciação

```
(define (begin? exp)
  (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))

(define (make-begin seq) (cons 'begin seq))

(define (eval-sequence exps env)
```

Avaliador

Sequenciação

```
(define (begin? exp)
  (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))

(define (make-begin seq) (cons 'begin seq))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
```

Avaliador

Sequenciação

```
(define (begin? exp)
  (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))

(define (make-begin seq) (cons 'begin seq))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
```

Avaliador

Sequenciação

```
(define (begin? exp)
  (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))

(define (make-begin seq) (cons 'begin seq))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

Avaliador

Expressões derivadas

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))


        ((if? exp) (eval-if exp env))





        ((begin? exp)
         (eval-sequence (begin-actions exp) env))





        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Expressões derivadas

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))


        ((if? exp) (eval-if exp env))




        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))



        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Expressões derivadas

```
(define (cond? exp)
```

Avaliador

Expressões derivadas

```
(define (cond? exp)
  (tagged-list? exp 'cond))
```

Avaliador

Expressões derivadas

```
(define (cond? exp)
  (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))
```

Avaliador

Expressões derivadas

```
(define (cond? exp)
  (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
```

Avaliador

Expressões derivadas

```
(define (cond? exp)
  (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))
```

Avaliador

Expressões derivadas

```
(define (cond? exp)
  (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))

(define (cond-predicate clause) (car clause))
```

Avaliador

Expressões derivadas

```
(define (cond? exp)
  (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))

(define (cond-predicate clause) (car clause))

(define (cond-actions clause) (cdr clause))
```

Avaliador

Expressões derivadas

```
(define (cond? exp)
  (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))

(define (cond-predicate clause) (car clause))

(define (cond-actions clause) (cdr clause))

(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                          ; no else clause
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                              ; no else clause
      (let ((first (car clauses))
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                            ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                            ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                           ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                          ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                          ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                          ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                                  ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                          ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                          ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))

(define (sequence->exp seq)
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                                ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))

(define (sequence->exp seq)
  (cond ((null? seq) seq)
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                          ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))

(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
```

Avaliador

Expressões derivadas

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                          ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))

(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
```

Avaliador

Aplicação de função

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))


        ((if? exp) (eval-if exp env))





        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))




        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Aplicação de função

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))


        ((if? exp) (eval-if exp env))




        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)


        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Aplicação de função

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))


        ((if? exp) (eval-if exp env))




        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)

        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Aplicação de função

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))


        ((if? exp) (eval-if exp env))





        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Aplicação de função

```
(define (application? exp)
```

Avaliador

Aplicação de função

```
(define (application? exp)
  (pair? exp))
```

Avaliador

Aplicação de função

```
(define (application? exp)
  (pair? exp))

(define (operator exp) (car exp))
```

Avaliador

Aplicação de função

```
(define (application? exp)
  (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))
```

Avaliador

Aplicação de função

```
(define (application? exp)
  (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))
```

Avaliador

Aplicação de função

```
(define (application? exp)
  (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))

(define (first-operand ops) (car ops))
```

Avaliador

Aplicação de função

```
(define (application? exp)
  (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))

(define (first-operand ops) (car ops))

(define (rest-operands ops) (cdr ops))
```

Avaliador

Aplicação de função

```
(define (application? exp)
  (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))

(define (first-operand ops) (car ops))

(define (rest-operands ops) (cdr ops))

(define (list-of-values exps env)
```

Avaliador

Aplicação de função

```
(define (application? exp)
  (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))

(define (first-operand ops) (car ops))

(define (rest-operands ops) (cdr ops))

(define (list-of-values exps env)
  (if (no-operands? exps)
```

Avaliador

Aplicação de função

```
(define (application? exp)
  (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))

(define (first-operand ops) (car ops))

(define (rest-operands ops) (cdr ops))

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
```

Avaliador

Aplicação de função

```
(define (application? exp)
  (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))

(define (first-operand ops) (car ops))

(define (rest-operands ops) (cdr ops))

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
```

Avaliador

Aplicação de função

```
(define (application? exp)
  (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))

(define (first-operand ops) (car ops))

(define (rest-operands ops) (cdr ops))

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

(define (eval-sequence exps env)
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
          (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

Avaliador

Aplicação de função

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
          (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
          (eval-sequence
            (procedure-body procedure)
            (extend-environment
              (procedure-parameters procedure)
              arguments
              (procedure-environment procedure))))
        (else
          (error
           "Unknown procedure type -- APPLY" procedure))))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

Avaliador

Definições

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))


        ((if? exp) (eval-if exp env))




        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Definições

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))

        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))




        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Definições

```
(define (definition? exp)
```

Avaliador

Definições

```
(define (definition? exp)
  (tagged-list? exp 'define))
```

Avaliador

Definições

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
```

Avaliador

Definições

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))
```

Avaliador

Definições

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
```

Avaliador

Definições

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)    ; formal parameters
                   (cddr exp))))  ; body
```

Avaliador

Definições

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)    ; formal parameters
                   (cddr exp))))  ; body

(define (eval-definition exp env)
```

Avaliador

Definições

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)   ; formal parameters
                   (cddr exp)))) ; body

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
```

Avaliador

Definições

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)    ; formal parameters
                   (cddr exp))))  ; body

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
```

Avaliador

Definições

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)   ; formal parameters
                   (cddr exp)))) ; body

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
```

Avaliador

Definições

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)   ; formal parameters
                   (cddr exp)))) ; body

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)
```

Avaliador

Definições

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)    ; formal parameters
                   (cddr exp))))  ; body

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)
```

Avaliador

Atribuições

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))

        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))




        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Atribuições

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))




        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Atribuições

```
(define (assignment? exp)
```

Avaliador

Atribuições

```
(define (assignment? exp)
  (tagged-list? exp 'set!))
```

Avaliador

Atribuições

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))
```

Avaliador

Atribuições

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))
```

Avaliador

Atribuições

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))

(define (eval-assignment exp env)
```

Avaliador

Atribuições

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
```

Avaliador

Atribuições

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
```

Avaliador

Atribuições

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env))
```

Avaliador

Atribuições

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)
```

Avaliador

Atribuições

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)
```

Avaliador

Variáveis

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)

        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))




        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Variáveis

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))




        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Variáveis

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))




        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Variáveis

```
(define (variable? exp) (symbol? exp))
```

Avaliador

Procedimentos

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))




        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Procedimentos

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)



        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Procedimentos

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)


        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Procedimentos

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)

        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Procedimentos

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Procedimentos

```
(define (lambda? exp) (tagged-list? exp 'lambda))
```

Avaliador

Procedimentos

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))
```

Avaliador

Procedimentos

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))
```

Avaliador

Procedimentos

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))

(define (make-procedure parameters body env)
```

Avaliador

Procedimentos

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
```

Avaliador

Procedimentos

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
```

Avaliador

Procedimentos

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
  (tagged-list? p 'procedure))
```

Avaliador

Procedimentos

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
  (tagged-list? p 'procedure))

(define (procedure-parameters p) (cadr p))
```

Avaliador

Procedimentos

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
  (tagged-list? p 'procedure))

(define (procedure-parameters p) (cadr p))

(define (procedure-body p) (caddr p))
```

Avaliador

Procedimentos

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
  (tagged-list? p 'procedure))

(define (procedure-parameters p) (cadr p))

(define (procedure-body p) (caddr p))

(define (procedure-environment p) (cadddr p))
```

Avaliador

Ambientes: Operações

- `(lookup-variable-value <var> <env>)`
- `(extend-environment <variables> <values> <base-env>)`
- `(define-variable! <var> <value> <env>)`
- `(set-variable-value! <var> <value> <env>)`

Ambientes: Implementação

- Um ambiente é uma sequência de enquadramentos.
- Cada enquadramento é uma tabela de associações.
- Cada associação é um par variável/valor.

Avaliador

Ambientes

```
(define (enclosing-environment env) (cdr env))
```

Avaliador

Ambientes

```
(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))
```

Avaliador

Ambientes

```
(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))

(define the-empty-environment '())
```

Avaliador

Ambientes

```
(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))

(define the-empty-environment '())

(define (make-frame variables values)
  (cons variables values))
```

Avaliador

Ambientes

```
(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))

(define the-empty-environment '())

(define (make-frame variables values)
  (cons variables values))

(define (frame-variables frame) (car frame))
```

Avaliador

Ambientes

```
(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))

(define the-empty-environment '())

(define (make-frame variables values)
  (cons variables values))

(define (frame-variables frame) (car frame))

(define (frame-values frame) (cdr frame))
```

Avaliador

Ambientes

```
(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))

(define the-empty-environment '())

(define (make-frame variables values)
  (cons variables values))

(define (frame-variables frame) (car frame))

(define (frame-values frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
```

Avaliador

Ambientes

```
(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))

(define the-empty-environment '())

(define (make-frame variables values)
  (cons variables values))

(define (frame-variables frame) (car frame))

(define (frame-values frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)









  (env-loop env))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)







    (if (eq? env the-empty-environment)





  (env-loop env))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)







    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)




  (env-loop env))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)




    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))


  (env-loop env))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)







    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)






    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)




    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))



    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))


    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (car vals))

    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (define-variable! var val env)
```

Avaliador

Ambientes

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
```

Avaliador

Ambientes

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)






    (scan (frame-variables frame)
          (frame-values frame))))
```

Avaliador

Ambientes

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)




    (scan (frame-variables frame)
          (frame-values frame))))
```

Avaliador

Ambientes

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
             (add-binding-to-frame! var val frame))



    (scan (frame-variables frame)
          (frame-values frame))))
```

Avaliador

Ambientes

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars))


    (scan (frame-variables frame)
          (frame-values frame))))
```

Avaliador

Ambientes

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
             (set-car! vals val))

    (scan (frame-variables frame)
          (frame-values frame))))
```

Avaliador

Ambientes

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

Avaliador

Ambientes

```
(define (set-variable-value! var val env)
```

Avaliador

Ambientes

```
(define (set-variable-value! var val env)
  (define (env-loop env)











  (env-loop env))
```

Avaliador

Ambientes

```
(define (set-variable-value! var val env)
  (define (env-loop env)







    (if (eq? env the-empty-environment)





  (env-loop env))
```

Avaliador

Ambientes

```
(define (set-variable-value! var val env)
  (define (env-loop env)




    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)



  (env-loop env))
```

Avaliador

Ambientes

```
(define (set-variable-value! var val env)
  (define (env-loop env)








    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))


  (env-loop env))
```

Avaliador

Ambientes

```
(define (set-variable-value! var val env)
  (define (env-loop env)







    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)







    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)




    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))



    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))


    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Avaliador

Ambientes

```
(define the-global-environment (setup-environment))
```

Avaliador

Ambientes

```
(define the-global-environment (setup-environment))

(define (setup-environment)
```

Avaliador

Ambientes

```
(define the-global-environment (setup-environment))

(define (setup-environment)
  (let ((initial-env
```

Avaliador

Ambientes

```
(define the-global-environment (setup-environment))

(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
```

Avaliador

Ambientes

```
(define the-global-environment (setup-environment))

(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
```

Avaliador

Ambientes

```
(define the-global-environment (setup-environment))

(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
    initial-env))
```

Avaliador

Ambientes

```
(define (apply procedure arguments)
```

Avaliador

Ambientes

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
```

Avaliador

Ambientes

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
```

Avaliador

Ambientes

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
```

Avaliador

Ambientes

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         ...)
```

Avaliador

Ambientes

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         ...)
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

Avaliador

Ambientes

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
          (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
          ...)
        (else
          (error
           "Unknown procedure type -- APPLY" procedure))))
```

Avaliador

Ambientes

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         ...)
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))



(define (primitive-procedure? proc)
```

Avaliador

Ambientes

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         ...)
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))



(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))
```

Avaliador

Ambientes

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         ...)
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))



(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

(define (primitive-implementation proc) (cadr proc))
```

Avaliador

Ambientes

```
(define primitive-procedures
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        <more primitives>))
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        <more primitives>))

(define (primitive-procedure-names)
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        <more primitives>))

(define (primitive-procedure-names)
  (map car
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        <more primitives>))

(define (primitive-procedure-names)
  (map car
       primitive-procedures))
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        <more primitives>))

(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        <more primitives>))

(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        <more primitives>))

(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        <more primitives>))

(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))

(define (apply-primitive-procedure proc args)
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        <more primitives>))

(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))

(define (apply-primitive-procedure proc args)
  (apply
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        <more primitives>))

(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))

(define (apply-primitive-procedure proc args)
  (apply
   (primitive-implementation proc) args))
```

Avaliador

Ambientes

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        <more primitives>))

(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))

(define (apply-primitive-procedure proc args)
  (apply
   (primitive-implementation proc) args))
```

Avaliador

Erro?

```
(define (apply-primitive-procedure proc args)
  (apply
   (primitive-implementation proc) args))
```

Avaliador

Erro?

```
(define (apply-primitive-procedure proc args)
  (apply
   (primitive-implementation proc) args))

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         ...)
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

Avaliador

Erro?

```
(define (apply-primitive-procedure proc args)
  (apply
   (primitive-implementation proc) args))

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         ...)
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

Avaliador

Eliminando o Erro

```
(define apply-in-underlying-scheme apply)
```

Avaliador

Eliminando o Erro

```
(define apply-in-underlying-scheme apply)

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         ...)
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

Avaliador

Eliminando o Erro

```
(define apply-in-underlying-scheme apply)

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         ...)
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))
```

Avaliador

Eliminando o Erro

```
(define apply-in-underlying-scheme apply)

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         ...)
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")

(define output-prompt ";;; M-Eval value:")
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")

(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")

(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")

(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")

(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")

(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")

(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output))))
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")

(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")

(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

(define (prompt-for-input string)
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")

(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")

(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
```

Avaliador

REPL

```
(define input-prompt ";;; M-Eval input:")

(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
  (newline) (display string) (newline))
```

Avaliador

REPL

```
(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))
```

Avaliador

Exemplo

```
(driver-loop)
```

Avaliador

Exemplo

```
(driver-loop)
;;; M-Eval input:
```

Avaliador

Exemplo

```
(driver-loop)
;;; M-Eval input:
(define (append x y)
```

Avaliador

Exemplo

```
(driver-loop)
;;; M-Eval input:
(define (append x y)
  (if (null? x)
```

Avaliador

Exemplo

```
(driver-loop)
;;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
```

Avaliador

Exemplo

```
(driver-loop)
;;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
```

Avaliador

Exemplo

```
(driver-loop)
;;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
```

Avaliador

Exemplo

```
(driver-loop)
;;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
;;; M-Eval value:
```

Avaliador

Exemplo

```
(driver-loop)
;;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
;;; M-Eval value:
ok
```

Avaliador

Exemplo

```
(driver-loop)
;;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
;;; M-Eval value:
ok
;;; M-Eval input:
```

Avaliador

Exemplo

```
(driver-loop)
;;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
;;; M-Eval value:
ok
;;; M-Eval input:
(append '(a b c) '(d e f))
```

Avaliador

Exemplo

```
(driver-loop)
;;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
;;; M-Eval value:
ok
;;; M-Eval input:
(append '(a b c) '(d e f))
;;; M-Eval value:
```

Avaliador

Exemplo

```
(driver-loop)
;;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
;;; M-Eval value:
ok
;;; M-Eval input:
(append '(a b c) '(d e f))
;;; M-Eval value:
(a b c d e f)
```

Avaliador

Eval

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Eval

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((current-environment? exp) env)
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Eval

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((current-environment? exp) env)
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Eval

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((current-environment? exp) env)
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Avaliador

Eval

```
(define (current-environment? exp)
  (tagged-list? exp 'current-environment))
```

Avaliador

Eval

```
(define (current-environment? exp)
  (tagged-list? exp 'current-environment))

(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
```

Avaliador

Eval

```
(define (current-environment? exp)
  (tagged-list? exp 'current-environment))

(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        (list 'eval eval)
```

Avaliador

Eval

```
(define (current-environment? exp)
  (tagged-list? exp 'current-environment))

(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        (list 'eval eval)
        <more primitives>))
```

Avaliador

Eval

```
(driver-loop)
```

Avaliador

Eval

```
(driver-loop)
;;; M-Eval input:
```

Avaliador

Eval

```
(driver-loop)
;;; M-Eval input:
(eval (cons '+ '(2 3)) (current-environment))
```

Avaliador

Eval

```
(driver-loop)
;;; M-Eval input:
(eval (cons '+ '(2 3)) (current-environment))
;;; M-Eval value:
```

Avaliador

Eval

```
(driver-loop)
;;; M-Eval input:
(eval (cons '+ '(2 3)) (current-environment))
;;; M-Eval value:
5
```